couchdb-python Documentation

Release 1.0

Dirkjan Ochtman

Feb 16, 2018

Contents

`couchdb` is Python package for working with CouchDB from Python code. It consists of the following main modules:

- `couchdb.client`: This is the client library for interfacing CouchDB servers. If you don't know where to start, this is likely to be what you're looking for.
- `couchdb.mapping`: This module provides advanced mapping between CouchDB JSON documents and Python objects.

Additionally, the `couchdb.view` module implements a view server for views written in Python.

There may also be more information on the project website.

CHAPTER 1

Documentation

1.1 Getting started with couchdb-python

Some snippets of code to get you started with writing code against CouchDB.

Starting off:

```
>>> import couchdb
>>> couch = couchdb.Server()
```

This gets you a Server object, representing a CouchDB server. By default, it assumes CouchDB is running on localhost:5984. If your CouchDB server is running elsewhere, set it up like this:

```
>>> couch = couchdb.Server('http://example.com:5984/')
```

You can also pass authentication credentials and/or use SSL:

```
>>> couch = couchdb.Server('https://username:password@host:port/')
```

You can create a new database from Python, or use an existing database:

```
>>> db = couch.create('test') # newly created
>>> db = couch['mydb'] # existing
```

After selecting a database, create a document and insert it into the db:

```
>>> doc = {'foo': 'bar'}
>>> db.save(doc)
('e0658cab843b59e63c8779a9a5000b01', '1-4c6114c65e295552ab1019e2b046b10e')
>>> doc
{'_rev': '1-4c6114c65e295552ab1019e2b046b10e', 'foo': 'bar', '_id':
↪'e0658cab843b59e63c8779a9a5000b01'}
```

The `save()` method returns the ID and "rev" for the newly created document. You can also set your own ID by including an `_id` item in the document.

Getting the document out again is easy:

```
>>> db['e0658cab843b59e63c8779a9a5000b01']
<Document 'e0658cab843b59e63c8779a9a5000b01'@'1-4c6114c65e295552ab1019e2b046b10e' {
↪'foo': 'bar'}>
```

To find all your documents, simply iterate over the database:

```
>>> for id in db:
...     print id
...
'e0658cab843b59e63c8779a9a5000b01'
```

Now we can clean up the test document and database we created:

```
>>> db.delete(doc)
>>> couch.delete('test')
```

1.2 Writing views in Python

The couchdb-python package comes with a view server to allow you to write views in Python instead of JavaScript. When couchdb-python is installed, it will install a script called couchpy that runs the view server. To enable this for your CouchDB server, add the following section to local.ini:

```
[query_servers]
python=/usr/bin/couchpy
```

After restarting CouchDB, the Futon view editor should show `python` in the language pull-down menu. Here's some sample view code to get you started:

```
def fun(doc):
    if 'date' in doc:
        yield doc['date'], doc
```

Note that the `map` function uses the Python `yield` keyword to emit values, where JavaScript views use an `emit()` function.

1.3 Basic CouchDB API: couchdb.client

Python client API for CouchDB.

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> doc_id, doc_rev = db.save({'type': 'Person', 'name': 'John Doe'})
>>> doc = db[doc_id]
>>> doc['type']
u'Person'
>>> doc['name']
u'John Doe'
>>> del db[doc.id]
>>> doc.id in db
False
```

```
>>> del server['python-tests']
```

1.3.1 Server

class couchdb.client.**Server**(*url='http://localhost:5984/'*, *full_commit=True*, *session=None*)

Representation of a CouchDB server.

```
>>> server = Server() # connects to the local_server
>>> remote_server = Server('http://example.com:5984/')
>>> secure_remote_server = Server('https://username:password@example.com:5984/')
```

This class behaves like a dictionary of databases. For example, to get a list of database names on the server, you can simply iterate over the server object.

New databases can be created using the *create* method:

```
>>> db = server.create('python-tests')
>>> db
<Database 'python-tests'>
```

You can access existing databases using item access, specifying the database name as the key:

```
>>> db = server['python-tests']
>>> db.name
'python-tests'
```

Databases can be deleted using a `del` statement:

```
>>> del server['python-tests']
```

add_user(*name*, *password*, *roles=None*)

Add regular user in authentication database.

Parameters

- **name** – name of regular user, normally user id
- **password** – password of regular user
- **roles** – roles of regular user

Returns (id, rev) tuple of the registered user

Return type *tuple*

config()

The configuration of the CouchDB server.

The configuration is represented as a nested dictionary of sections and options from the configuration files of the server, or the default values for options that are not explicitly configured.

Return type *dict*

create(*name*)

Create a new database with the given name.

Parameters **name** – the name of the database

Returns a *Database* object representing the created database

Return type *Database*

Raises `PreconditionFailed` – if a database with that name already exists

`delete`(*name*)
Delete the database with the specified name.

Parameters `name` – the name of the database

Raises `ResourceNotFound` – if a database with that name does not exist

Since 0.6

`login`(*name*, *password*)
Login regular user in couch db

Parameters

- `name` – name of regular user, normally user id
- `password` – password of regular user

Returns authentication token

`logout`(*token*)
Logout regular user in couch db

Parameters `token` – token of login user

Returns True if successfully logout

Return type bool

`remove_user`(*name*)
Remove regular user in authentication database.

Parameters `name` – name of regular user, normally user id

`replicate`(*source*, *target*, ***options*)
Replicate changes from the source database to the target database.

Parameters

- `source` – URL of the source database
- `target` – URL of the target database
- `options` – optional replication args, e.g. continuous=True

`stats`(*name=None*)
Server statistics.

Parameters `name` – name of single statistic, e.g. httpd/requests (None – return all statistics)

`tasks`()
A list of tasks currently active on the server.

`uuids`(*count=None*)
Retrieve a batch of uuids

Parameters `count` – a number of uuids to fetch (None – get as many as the server sends)

Returns a list of uuids

`verify_token`(*token*)
Verify user token

Parameters `token` – authentication token

Returns True if authenticated ok

Return type bool

version()
The version string of the CouchDB server.

Note that this results in a request being made, and can also be used to check for the availability of the server.

Return type *unicode*

version_info()
The version of the CouchDB server as a tuple of ints.

Note that this results in a request being made only at the first call. Afterwards the result will be cached.

Return type *tuple(int, int, int)*

1.3.2 Database

class couchdb.client.**Database**(*url*, *name=None*, *session=None*)
Representation of a database on a CouchDB server.

```
>>> server = Server()
>>> db = server.create('python-tests')
```

New documents can be added to the database using the *save()* method:

```
>>> doc_id, doc_rev = db.save({'type': 'Person', 'name': 'John Doe'})
```

This class provides a dictionary-like interface to databases: documents are retrieved by their ID using item access

```
>>> doc = db[doc_id]
>>> doc
<Document u'...'@... {...}>
```

Documents are represented as instances of the *Row* class, which is basically just a normal dictionary with the additional attributes `id` and `rev`:

```
>>> doc.id, doc.rev
(u'...', ...)
>>> doc['type']
u'Person'
>>> doc['name']
u'John Doe'
```

To update an existing document, you use item access, too:

```
>>> doc['name'] = 'Mary Jane'
>>> db[doc.id] = doc
```

The *save()* method creates a document with a random ID generated by CouchDB (which is not recommended). If you want to explicitly specify the ID, you'd use item access just as with updating:

```
>>> db['JohnDoe'] = {'type': 'person', 'name': 'John Doe'}
>>> 'JohnDoe' in db
True
>>> len(db)
2
```

```
>>> del server['python-tests']
```

If you need to connect to a database with an unverified or self-signed SSL certificate, you can re-initialize your ConnectionPool as follows (only applicable for Python 2.7.9+):

```
>>> db.resource.session.disable_ssl_verification()
```

changes(***opts*)
Retrieve a changes feed from the database.

Parameters **opts** – optional query string parameters

Returns an iterable over change notification dicts

cleanup()
Clean up old design document indexes.

Remove all unused index files from the database storage area.

Returns a boolean to indicate successful cleanup initiation

Return type *bool*

commit()
If the server is configured to delay commits, or previous requests used the special `X-Couch-Full-Commit: false` header to disable immediate commits, this method can be used to ensure that any non-committed changes are committed to physical storage.

compact(*ddoc=None*)
Compact the database or a design document's index.

Without an argument, this will try to prune all old revisions from the database. With an argument, it will compact the index cache for all views in the design document specified.

Returns a boolean to indicate whether the compaction was initiated successfully

Return type *bool*

copy(*src*, *dest*)
Copy the given document to create a new document.

Parameters

- **src** – the ID of the document to copy, or a dictionary or *Document* object representing the source document.
- **dest** – either the destination document ID as string, or a dictionary or *Document* instance of the document that should be overwritten.

Returns the new revision of the destination document

Return type *str*

Since 0.6

create(*data*)
Create a new document in the database with a random ID that is generated by the server.

Note that it is generally better to avoid the *create()* method and instead generate document IDs on the client side. This is due to the fact that the underlying HTTP `POST` method is not idempotent, and an automatic retry due to a problem somewhere on the networking stack may cause multiple documents being created in the database.

To avoid such problems you can generate a UUID on the client side. Python (since version 2.5) comes with a `uuid` module that can be used for this:

```
from uuid import uuid4
doc_id = uuid4().hex
db[doc_id] = {'type': 'person', 'name': 'John Doe'}
```

Parameters **data** – the data to store in the document

Returns the ID of the created document

Return type *unicode*

delete(*doc*)

Delete the given document from the database.

Use this method in preference over `__del__` to ensure you're deleting the revision that you had previously retrieved. In the case the document has been updated since it was retrieved, this method will raise a *ResourceConflict* exception.

```
>>> server = Server()
>>> db = server.create('python-tests')
```

```
>>> doc = dict(type='Person', name='John Doe')
>>> db['johndoe'] = doc
>>> doc2 = db['johndoe']
>>> doc2['age'] = 42
>>> db['johndoe'] = doc2
>>> db.delete(doc)
Traceback (most recent call last):
  ...
ResourceConflict: (u'conflict', u'Document update conflict.')
```

```
>>> del server['python-tests']
```

Parameters **doc** – a dictionary or *Document* object holding the document data

Raises **ResourceConflict** – if the document was updated in the database

Since 0.4.1

delete_attachment(*doc, filename*)

Delete the specified attachment.

Note that the provided *doc* is required to have a `_rev` field. Thus, if the *doc* is based on a view row, the view row would need to include the `_rev` field.

Parameters

- **doc** – the dictionary or *Document* object representing the document that the attachment belongs to
- **filename** – the name of the attachment file

Since 0.4.1

explain(*mango_query*)
Explain a mango find-query.

Note: only available for CouchDB version >= 2.0.0

More information on the *mango_query* structure can be found here: http://docs.couchdb.org/en/master/api/database/find.html#db-explain

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> db['johndoe'] = dict(type='Person', name='John Doe')
>>> db['maryjane'] = dict(type='Person', name='Mary Jane')
>>> db['gotham'] = dict(type='City', name='Gotham City')
>>> mango = {'selector': {'type': 'Person'}, 'fields': ['name']}
>>> db.explain(mango)
{...}
>>> del server['python-tests']
```

Parameters **mango_query** – a *dict* describing criteria used to select documents

Returns the query results as a list of *Document* (or whatever *wrapper* returns)

Return type *dict*

find(*mango_query*, *wrapper=None*)
Execute a mango find-query against the database.

Note: only available for CouchDB version >= 2.0.0

More information on the *mango_query* structure can be found here: http://docs.couchdb.org/en/master/api/database/find.html#find-selectors

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> db['johndoe'] = dict(type='Person', name='John Doe')
>>> db['maryjane'] = dict(type='Person', name='Mary Jane')
>>> db['gotham'] = dict(type='City', name='Gotham City')
>>> mango = {'selector': {'type': 'Person'},
...          'fields': ['name'],
...          'sort':[{'name': 'asc'}]}
>>> for row in db.find(mango):
...     print(row['name'])
John Doe
Mary Jane
>>> del server['python-tests']
```

Parameters

- **mango_query** – a dictionary describing criteria used to select documents
- **wrapper** – an optional callable that should be used to wrap the resulting documents

Returns the query results as a list of *Document* (or whatever *wrapper* returns)

get(*id*, *default=None*, ***options*)
Return the document with the specified ID.

Parameters

- **`id`** – the document ID
- **`default`** – the default value to return when the document is not found

Returns a *Row* object representing the requested document, or *None* if no document with the ID was found

Return type *Document*

`get_attachment`(*id_or_doc*, *filename*, *default=None*)
Return an attachment from the specified doc id and filename.

Parameters

- **`id_or_doc`** – either a document ID or a dictionary or *Document* object representing the document that the attachment belongs to
- **`filename`** – the name of the attachment file
- **`default`** – default value to return when the document or attachment is not found

Returns a file-like object with read and close methods, or the value of the *default* argument if the attachment is not found

Since 0.4.1

`index`()
Get an object to manage the database indexes.

Returns an *Indexes* object to manage the databes indexes

Return type *Indexes*

`info`(*ddoc=None*)
Return information about the database or design document as a dictionary.

Without an argument, returns database information. With an argument, return information for the given design document.

The returned dictionary exactly corresponds to the JSON response to a `GET` request on the database or design document's info URI.

Returns a dictionary of database properties

Return type `dict`

Since 0.4

`iterview`(*name*, *batch*, *wrapper=None*, ***options*)
Iterate the rows in a view, fetching rows in batches and yielding one row at a time.

Since the view's rows are fetched in batches any rows emitted for documents added, changed or deleted between requests may be missed or repeated.

Parameters

- **`name`** – the name of the view; for custom views, use the format `design_docid/viewname`, that is, the document ID of the design document and the name of the view, separated by a slash.
- **`batch`** – number of rows to fetch per HTTP request.
- **`wrapper`** – an optional callable that should be used to wrap the result rows
- **`options`** – optional query string parameters

Returns row generator

list (*name*, *view*, ***options*)
Format a view using a 'list' function.

Parameters

- **name** – the name of the list function in the format `designdoc/listname`
- **view** – the name of the view in the format `designdoc/viewname`
- **options** – optional query string parameters

Returns (headers, body) tuple, where headers is a dict of headers returned from the list function and body is a readable file-like instance

name
The name of the database.

Note that this may require a request to the server unless the name has already been cached by the *info()* method.

Return type basestring

purge (*docs*)
Perform purging (complete removing) of the given documents.

Uses a single HTTP request to purge all given documents. Purged documents do not leave any meta-data in the storage and are not replicated.

put_attachment (*doc*, *content*, *filename=None*, *content_type=None*)
Create or replace an attachment.

Note that the provided *doc* is required to have a `_rev` field. Thus, if the *doc* is based on a view row, the view row would need to include the `_rev` field.

Parameters

- **doc** – the dictionary or *Document* object representing the document that the attachment should be added to
- **content** – the content to upload, either a file-like object or a string
- **filename** – the name of the attachment file; if omitted, this function tries to get the filename from the file-like object passed as the *content* argument value
- **content_type** – content type of the attachment; if omitted, the MIME type is guessed based on the file name extension

Since 0.4.1

query (*map_fun*, *reduce_fun=None*, *language='javascript'*, *wrapper=None*, ***options*)
Execute an ad-hoc query (a "temp view") against the database.

Note: not supported for CouchDB version >= 2.0.0

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> db['johndoe'] = dict(type='Person', name='John Doe')
>>> db['maryjane'] = dict(type='Person', name='Mary Jane')
>>> db['gotham'] = dict(type='City', name='Gotham City')
>>> map_fun = '''function(doc) {
...     if (doc.type == 'Person')
...         emit(doc.name, null);
... }'''
>>> for row in db.query(map_fun):
...     print(row.key)
```

```
John Doe
Mary Jane
```

```
>>> for row in db.query(map_fun, descending=True):
...     print(row.key)
Mary Jane
John Doe
```

```
>>> for row in db.query(map_fun, key='John Doe'):
...     print(row.key)
John Doe
```

```
>>> del server['python-tests']
```

Parameters

- **map_fun** – the code of the map function
- **reduce_fun** – the code of the reduce function (optional)
- **language** – the language of the functions, to determine which view server to use
- **wrapper** – an optional callable that should be used to wrap the result rows
- **options** – optional query string parameters

Returns the view results

Return type *ViewResults*

revisions(*id*, ***options*)
Return all available revisions of the given document.

Parameters **id** – the document ID

Returns an iterator over Document objects, each a different revision, in reverse chronological order, if any were found

save(*doc*, ***options*)
Create a new document or update an existing document.

If doc has no _id then the server will allocate a random ID and a new document will be created. Otherwise the doc's _id will be used to identify the document to create or update. Trying to update an existing document with an incorrect _rev will raise a ResourceConflict exception.

Note that it is generally better to avoid saving documents with no _id and instead generate document IDs on the client side. This is due to the fact that the underlying HTTP `POST` method is not idempotent, and an automatic retry due to a problem somewhere on the networking stack may cause multiple documents being created in the database.

To avoid such problems you can generate a UUID on the client side. Python (since version 2.5) comes with a `uuid` module that can be used for this:

```
from uuid import uuid4
doc = {'_id': uuid4().hex, 'type': 'person', 'name': 'John Doe'}
db.save(doc)
```

Parameters

- **doc** – the document to store

- **options** – optional args, e.g. batch='ok'

Returns (id, rev) tuple of the save document

Return type *tuple*

show (*name*, *docid=None*, ***options*)
Call a 'show' function.

Parameters

- **name** – the name of the show function in the format `designdoc/showname`
- **docid** – optional ID of a document to pass to the show function.
- **options** – optional query string parameters

Returns (headers, body) tuple, where headers is a dict of headers returned from the show function and body is a readable file-like instance

update (*documents*, ***options*)
Perform a bulk update or insertion of the given documents using a single HTTP request.

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> for doc in db.update([
...     Document(type='Person', name='John Doe'),
...     Document(type='Person', name='Mary Jane'),
...     Document(type='City', name='Gotham City')
... ]):
...     print(repr(doc))
(True, u'...', u'...')
(True, u'...', u'...')
(True, u'...', u'...')
```

```
>>> del server['python-tests']
```

The return value of this method is a list containing a tuple for every element in the *documents* sequence. Each tuple is of the form `(success, docid, rev_or_exc)`, where `success` is a boolean indicating whether the update succeeded, `docid` is the ID of the document, and `rev_or_exc` is either the new document revision, or an exception instance (e.g. *ResourceConflict*) if the update failed.

If an object in the documents list is not a dictionary, this method looks for an `items()` method that can be used to convert the object to a dictionary. Effectively this means you can also use this method with *mapping.Document* objects.

Parameters **documents** – a sequence of dictionaries or *Document* objects, or objects providing a `items()` method that can be used to convert them to a dictionary

Returns an iterable over the resulting documents

Return type `list`

Since version 0.2

update_doc (*name*, *docid=None*, ***options*)
Calls server side update handler.

Parameters

- **name** – the name of the update handler function in the format `designdoc/updatename`.

- **`docid`** – optional ID of a document to pass to the update handler.
- **`options`** – additional (optional) params to pass to the underlying http resource handler, including `headers`, `body`, and ``` `path` ```. Other arguments will be treated as query string params. See `couchdb.http.Resource`

Returns (headers, body) tuple, where headers is a dict of headers returned from the list function and body is a readable file-like instance

`view`(*name*, *wrapper=None*, ***options*)

Execute a predefined view.

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> db['gotham'] = dict(type='City', name='Gotham City')
```

```
>>> for row in db.view('_all_docs'):
...     print(row.id)
gotham
```

```
>>> del server['python-tests']
```

Parameters

- **`name`** – the name of the view; for custom views, use the format `design_docid/viewname`, that is, the document ID of the design document and the name of the view, separated by a slash
- **`wrapper`** – an optional callable that should be used to wrap the result rows
- **`options`** – optional query string parameters

Returns the view results

Return type *ViewResults*

1.3.3 Document

class `couchdb.client.`**`Document`**

Representation of a document in the database.

This is basically just a dictionary with the two additional properties *id* and *rev*, which contain the document ID and revision, respectively.

`id`

The document ID.

Return type basestring

`rev`

The document revision.

Return type basestring

1.3.4 ViewResults

class `couchdb.client.`**`ViewResults`**(*view*, *options*)

Representation of a parameterized view (either permanent or temporary) and the results it produces.

This class allows the specification of `key`, `startkey`, and `endkey` options using Python slice notation.

```
>>> server = Server()
>>> db = server.create('python-tests')
>>> db['johndoe'] = dict(type='Person', name='John Doe')
>>> db['maryjane'] = dict(type='Person', name='Mary Jane')
>>> db['gotham'] = dict(type='City', name='Gotham City')
>>> map_fun = '''function(doc) {
...     emit([doc.type, doc.name], doc.name);
... }'''
>>> results = db.query(map_fun)
```

At this point, the view has not actually been accessed yet. It is accessed as soon as it is iterated over, its length is requested, or one of its *rows*, *total_rows*, or *offset* properties are accessed:

```
>>> len(results)
3
```

You can use slices to apply `startkey` and/or `endkey` options to the view:

```
>>> people = results[['Person']:['Person','ZZZZ']]
>>> for person in people:
...     print(person.value)
John Doe
Mary Jane
>>> people.total_rows, people.offset
(3, 1)
```

Use plain indexed notation (without a slice) to apply the `key` option. Note that as CouchDB makes no claim that keys are unique in a view, this can still return multiple rows:

```
>>> list(results[['City', 'Gotham City']])
[<Row id=u'gotham', key=[u'City', u'Gotham City'], value=u'Gotham City'>]
```

```
>>> del server['python-tests']
```

offset
The offset of the results from the first row in the view.

This value is 0 for reduce views.

Return type *int*

rows
The list of rows returned by the view.

Return type *list*

total_rows
The total number of rows in this view.

This value is *None* for reduce views.

Return type *int* or `NoneType` for reduce views

update_seq
The database update sequence that the view reflects.

The update sequence is included in the view result only when it is explicitly requested using the *update_seq=true* query option. Otherwise, the value is None.

Return type *int* or *NoneType* depending on the query options

1.3.5 Row

class `couchdb.client.`**`Row`**

Representation of a row as returned by database views.

`doc`

The associated document for the row. This is only present when the view was accessed with `include_docs=True` as a query parameter, otherwise this property will be *None*.

`id`

The associated Document ID if it exists. Returns *None* when it doesn't (reduce results).

1.4 Mapping CouchDB documents to Python objects: couchdb.mapping

Mapping from raw JSON data structures to Python objects and vice versa.

```
>>> from couchdb import Server
>>> server = Server()
>>> db = server.create('python-tests')
```

To define a document mapping, you declare a Python class inherited from *Document*, and add any number of *Field* attributes:

```
>>> from datetime import datetime
>>> from couchdb.mapping import Document, TextField, IntegerField, DateTimeField
>>> class Person(Document):
...     name = TextField()
...     age = IntegerField()
...     added = DateTimeField(default=datetime.now)
>>> person = Person(name='John Doe', age=42)
>>> person.store(db)
<Person ...>
>>> person.age
42
```

You can then load the data from the CouchDB server through your *Document* subclass, and conveniently access all attributes:

```
>>> person = Person.load(db, person.id)
>>> old_rev = person.rev
>>> person.name
u'John Doe'
>>> person.age
42
>>> person.added
datetime.datetime(...)
```

To update a document, simply set the attributes, and then call the `store()` method:

```
>>> person.name = 'John R. Doe'
>>> person.store(db)
<Person ...>
```

If you retrieve the document from the server again, you should be getting the updated data:

```
>>> person = Person.load(db, person.id)
>>> person.name
u'John R. Doe'
>>> person.rev != old_rev
True
```

```
>>> del server['python-tests']
```

1.4.1 Field types

class couchdb.mapping.**TextField**(*name=None*, *default=None*)
Mapping field for string values.

class couchdb.mapping.**FloatField**(*name=None*, *default=None*)
Mapping field for float values.

class couchdb.mapping.**IntegerField**(*name=None*, *default=None*)
Mapping field for integer values.

class couchdb.mapping.**LongField**(*name=None*, *default=None*)
Mapping field for long integer values.

class couchdb.mapping.**BooleanField**(*name=None*, *default=None*)
Mapping field for boolean values.

class couchdb.mapping.**DecimalField**(*name=None*, *default=None*)
Mapping field for decimal values.

class couchdb.mapping.**DateField**(*name=None*, *default=None*)
Mapping field for storing dates.

```
>>> field = DateField()
>>> field._to_python('2007-04-01')
datetime.date(2007, 4, 1)
>>> field._to_json(date(2007, 4, 1))
'2007-04-01'
>>> field._to_json(datetime(2007, 4, 1, 15, 30))
'2007-04-01'
```

class couchdb.mapping.**DateTimeField**(*name=None*, *default=None*)
Mapping field for storing date/time values.

```
>>> field = DateTimeField()
>>> field._to_python('2007-04-01T15:30:00Z')
datetime.datetime(2007, 4, 1, 15, 30)
>>> field._to_python('2007-04-01T15:30:00.009876Z')
datetime.datetime(2007, 4, 1, 15, 30, 0, 9876)
>>> field._to_json(datetime(2007, 4, 1, 15, 30, 0))
'2007-04-01T15:30:00Z'
>>> field._to_json(datetime(2007, 4, 1, 15, 30, 0, 9876))
'2007-04-01T15:30:00.009876Z'
>>> field._to_json(date(2007, 4, 1))
'2007-04-01T00:00:00Z'
```

class couchdb.mapping.**DictField**(*mapping=None*, *name=None*, *default=None*)
Field type for nested dictionaries.

```
>>> from couchdb import Server
>>> server = Server()
>>> db = server.create('python-tests')
```

```
>>> class Post(Document):
...     title = TextField()
...     content = TextField()
...     author = DictField(Mapping.build(
...         name = TextField(),
...         email = TextField()
...     ))
...     extra = DictField()
```

```
>>> post = Post(
...     title='Foo bar',
...     author=dict(name='John Doe',
...                 email='john@doe.com'),
...     extra=dict(foo='bar'),
... )
>>> post.store(db)
<Post ...>
>>> post = Post.load(db, post.id)
>>> post.author.name
u'John Doe'
>>> post.author.email
u'john@doe.com'
>>> post.extra
{u'foo': u'bar'}
```

```
>>> del server['python-tests']
```

class couchdb.mapping.**ListField**(*field*, *name=None*, *default=None*)

Field type for sequences of other fields.

```
>>> from couchdb import Server
>>> server = Server()
>>> db = server.create('python-tests')
```

```
>>> class Post(Document):
...     title = TextField()
...     content = TextField()
...     pubdate = DateTimeField(default=datetime.now)
...     comments = ListField(DictField(Mapping.build(
...         author = TextField(),
...         content = TextField(),
...         time = DateTimeField()
...     )))
```

```
>>> post = Post(title='Foo bar')
>>> post.comments.append(author='myself', content='Bla bla',
...                      time=datetime.now())
>>> len(post.comments)
1
>>> post.store(db)
<Post ...>
>>> post = Post.load(db, post.id)
>>> comment = post.comments[0]
>>> comment['author']
u'myself'
>>> comment['content']
u'Bla bla'
>>> comment['time']
u'...T...Z'
```

```
>>> del server['python-tests']
```

class `couchdb.mapping.`**`ViewField`**(*design*, *map_fun*, *reduce_fun=None*, *name=None*, *language='javascript'*, *wrapper=<object object>*, **defaults)

Descriptor that can be used to bind a view definition to a property of a *Document* class.

```
>>> class Person(Document):
...     name = TextField()
...     age = IntegerField()
...     by_name = ViewField('people', '''\
...         function(doc) {
...             emit(doc.name, doc);
...         }''')
>>> Person.by_name
<ViewDefinition '_design/people/_view/by_name'>
```

```
>>> print(Person.by_name.map_fun)
function(doc) {
    emit(doc.name, doc);
}
```

That property can be used as a function, which will execute the view.

```
>>> from couchdb import Database
>>> db = Database('python-tests')
```

```
>>> Person.by_name(db, count=3)
<ViewResults <PermanentView '_design/people/_view/by_name'> {'count': 3}>
```

The results produced by the view are automatically wrapped in the *Document* subclass the descriptor is bound to. In this example, it would return instances of the *Person* class. But please note that this requires the values of the view results to be dictionaries that can be mapped to the mapping defined by the containing *Document* class. Alternatively, the `include_docs` query option can be used to inline the actual documents in the view results, which will then be used instead of the values.

If you use Python view functions, this class can also be used as a decorator:

```
>>> class Person(Document):
...     name = TextField()
...     age = IntegerField()
...
...     @ViewField.define('people')
...     def by_name(doc):
...         yield doc['name'], doc
```

```
>>> Person.by_name
<ViewDefinition '_design/people/_view/by_name'>
```

```
>>> print(Person.by_name.map_fun)
def by_name(doc):
    yield doc['name'], doc
```

1.5 Changes

1.5.1 Version 1.2 (2018-02-09)

- Fixed some issues relating to usage with Python 3
- Remove support for Python 2.6 and 3.x with x < 4
- Fix logging response in query server (fixes #321)
- Fix HTTP authentication password encoding (fixes #302)
- Add missing `http.Forbidden` error (fixes #305)
- Show `doc` property on `Row` string representation
- Add methods for mango queries and indexes
- Allow mango filters in `_changes` API

1.5.2 Version 1.1 (2016-08-05)

- Add script to load design documents from disk
- Add methods on `Server` for user/session management
- Add microseconds support for DateTimeFields
- Handle changes feed as emitted by CouchBase (fixes #289)
- Support Python 3 in `couchdb-dump` script (fixes #296)
- Expand relative URLs from Location headers (fixes #287)
- Correctly handle `_rev` fields in mapped documents (fixes #278)

1.5.3 Version 1.0.1 (2016-03-12)

- Make sure connections are correctly closed on GAE (fixes #224)
- Correctly join path parts in replicate script (fixes #269)
- Fix id and rev for some special documents
- Make it possible to disable SSL verification

1.5.4 Version 1.0 (2014-11-16)

- Many smaller Python 3 compatibility issues have been fixed
- Improve handling of binary attachments in the `couchdb-dump` tool
- Added testing via tox and support for Travis CI

1.5.5 Version 0.10 (2014-07-15)

- Now compatible with Python 2.7, 3.3 and 3.4
- Added batch processing for the `couchdb-dump` tool
- A very basic API to access the `_security` object
- A way to access the `update_seq` value on view results

1.5.6 Version 0.9 (2013-04-25)

- Don't validate database names on the client side. This means some methods dealing with database names can return different exceptions than before.
- Use HTTP socket more efficiently to avoid the Nagle algorithm, greatly improving performace. Note: add the `{nodelay, true}` option to the CouchDB server's httpd/socket_options config.
- Add support for show and list functions.
- Add support for calling update handlers.
- Add support for purging documents.
- Add `iterview()` for more efficient iteration over large view results.
- Add view cleanup API.
- Enhance `Server.stats()` to optionally retrieve a single set of statistics.
- Implement `Session` timeouts.
- Add `error` property to `Row` objects.
- Add `default=None` arg to `mapping.Document.get()` to make it a little more dict-like.
- Enhance `Database.info()` so it can also be used to get info for a design doc.
- Add view definition options, e.g. collation.
- Fix support for authentication in dump/load tools.
- Support non-ASCII document IDs in serialization format.
- Protect `ResponseBody` from being iterated/closed multiple times.
- Rename iteration method for ResponseBody chunks to `iterchunks()` to prevent usage for non-chunked responses.
- JSON encoding exceptions are no longer masked, resulting in better error messages.
- `cjson` support is now deprecated.
- Fix `Row.value` and `Row.__repr__` to never raise exceptions.
- Fix Python view server's reduce to handle empty map results list.
- Use locale-independent timestamp identifiers for HTTP cache.
- Don't require setuptools/distribute to install the core package. (Still needed to install the console scripts.)

1.5.7 Version 0.8 (Aug 13, 2010)

- The couchdb-replicate script has changed from being a poor man's version of continuous replication (predating it) to being a simple script to help kick off replication jobs across databases and servers.
- Reinclude all http exception types in the 'couchdb' package's scope.
- Replaced epydoc API docs by more extensive Sphinx-based documentation.
- Request retries schedule and frequency are now customizable.
- Allow more kinds of request errors to trigger a retry.
- Improve wrapping of view results.
- Added a `uuids()` method to the `client.Server` class (issue 122).
- Tested with CouchDB 0.10 - 1.0 (and Python 2.4 - 2.7).

1.5.8 Version 0.7.0 (Apr 15, 2010)

- Breaking change: the dependency on `httplib2` has been replaced by an internal `couchdb.http` library. This changes the API in several places. Most importantly, `resource.request()` now returns a 3-member tuple.
- Breaking change: `couchdb.schema` has been renamed to `couchdb.mapping`. This better reflects what is actually provided. Classes inside `couchdb.mapping` have been similarly renamed (e.g. `Schema` -> `Mapping`).
- Breaking change: `couchdb.schema.View` has been renamed to `couchdb.mapping.ViewField`, in order to help distinguish it from `couchdb.client.View`.
- Breaking change: the `client.Server` properties `version` and `config` have become methods in order to improve API consistency.
- Prevent `schema.ListField` objects from sharing the same default (issue 107).
- Added a `changes()` method to the `client.Database` class (issue 103).
- Added an optional argument to the 'Database.compact'' method to enable view compaction (the rest of issue 37).

1.5.9 Version 0.6.1 (Dec 14, 2009)

- Compatible with CouchDB 0.9.x and 0.10.x.
- Removed debugging statement from `json` module (issue 82).
- Fixed a few bugs resulting from typos.
- Added a `replicate()` method to the `client.Server` class (issue 61).
- Honor the boundary argument in the dump script code (issue 100).
- Added a `stats()` method to the `client.Server` class.
- Added a `tasks()` method to the `client.Server` class.
- Allow slashes in path components passed to the uri function (issue 96).
- `schema.DictField` objects now have a separate backing dictionary for each instance of their `schema.Document` (issue 101).

- `schema.ListField` proxy objects now have a more consistent (though somewhat slower) `count()` method (issue 91).
- `schema.ListField` objects now have correct behavior for slicing operations and the `pop()` method (issue 92).
- Added a `revisions()` method to the Database class (issue 99).
- Make sure we always return UTF-8 from the view server (issue 81).

1.5.10 Version 0.6 (Jul 2, 2009)

- Compatible with CouchDB 0.9.x.
- `schema.DictField` instances no longer need to be bound to a `Schema` (issue 51).
- Added a `config` property to the `client.Server` class (issue 67).
- Added a `compact()` method to the `client.Database` class (issue 37).
- Changed the `update()` method of the `client.Database` class to simplify the handling of errors. The method now returns a list of `(success, docid, rev_or_exc)` tuples. See the docstring of that method for the details.
- `schema.ListField` proxy objects now support the `__contains__()` and `index()` methods (issue 77).
- The results of the `query()` and `view()` methods in the `schema.Document` class are now properly wrapped in objects of the class if the `include_docs` option is set (issue 76).
- Removed the `eager` option on the `query()` and `view()` methods of `schema.Document`. Use the `include_docs` option instead, which doesn't require an additional request per document.
- Added a `copy()` method to the `client.Database` class, which translates to a HTTP COPY request (issue 74).
- Accessing a non-existing database through `Server.__getitem__` now throws a `ResourceNotFound` exception as advertised (issue 41).
- Added a `delete()` method to the `client.Server` class for consistency (issue 64).
- The `couchdb-dump` tool now operates in a streaming fashion, writing one document at a time to the resulting MIME multipart file (issue 58).
- It is now possible to explicitly set the JSON module that should be used for decoding/encoding JSON data. The currently available choices are `simplejson`, `cjson`, and `json` (the standard library module). It is also possible to use custom decoding/encoding functions.
- Add logging to the Python view server. It can now be configured to log to a given file or the standard error stream, and the log level can be set debug to see all communication between CouchDB and the view server (issue 55).

1.5.11 Version 0.5 (Nov 29, 2008)

- `schema.Document` objects can now be used in the documents list passed to `client.Database.update()`.
- `Server.__contains__()` and `Database.__contains__()` now use the HTTP HEAD method to avoid unnecessary transmission of data. `Database.__del__()` also uses HEAD to determine the latest revision of the document.

- The `Database` class now has a method `delete()` that takes a document dictionary as parameter. This method should be used in preference to __`del`__ as it allow conflict detection and handling.
- Added `cache` and `timeout` arguments to the `client.Server` initializer.
- The `Database` class now provides methods for deleting, retrieving, and updating attachments.
- The Python view server now exposes a `log()` function to map and reduce functions (issue 21).
- Handling of the rereduce stage in the Python view server has been fixed.
- The `Server` and `Database` classes now implement the __`nonzero`__ hook so that they produce sensible results in boolean conditions.
- The client module will now reattempt a request that failed with a “connection reset by peer” error.
- inf/nan values now raise a `ValueError` on the client side instead of triggering an internal server error (issue 31).
- Added a new `couchdb.design` module that provides functionality for managing views in design documents, so that they can be defined in the Python application code, and the design documents actually stored in the database can be kept in sync with the definitions in the code.
- The `include_docs` option for CouchDB views is now supported by the new `doc` property of row instances in view results. Thanks to Paul Davis for the patch (issue 33).
- The `keys` option for views is now supported (issue 35).

1.5.12 Version 0.4 (Jun 28, 2008)

- Updated for compatibility with CouchDB 0.8.0
- Added command-line scripts for importing/exporting databases.
- The `Database.update()` function will now actually perform the `POST` request even when you do not iterate over the results (issue 5).
- The `_view` prefix can now be omitted when specifying view names.

1.5.13 Version 0.3 (Feb 6, 2008)

- The `schema.Document` class now has a `view()` method that can be used to execute a CouchDB view and map the result rows back to objects of that schema.
- The test suite now uses the new default port of CouchDB, 5984.
- Views now return proxy objects to which you can apply slice syntax for “key”, “startkey”, and “endkey” filtering.
- Add a `query()` classmethod to the `Document` class.

1.5.14 Version 0.2 (Nov 21, 2007)

- Added __len__ and __iter__ to the `schema.Schema` class to iterate over and get the number of items in a document or compound field.
- The “version” property of client.Server now returns a plain string instead of a tuple of ints.
- The client library now identifies itself with a meaningful User-Agent string.
- `schema.Document.store()` now returns the document object instance, instead of just the document ID.

- The string representation of `schema.Document` objects is now more comprehensive.
- Only the view parameters "key", "startkey", and "endkey" are JSON encoded, anything else is left alone.
- Slashes in document IDs are now URL-quoted until CouchDB supports them.
- Allow the content-type to be passed for temp views via `client.Database.query()` so that view languages other than Javascript can be used.
- Added `client.Database.update()` method to bulk insert/update documents in a database.
- The view-server script wrapper has been renamed to `couchpy`.
- `couchpy` now supports `--help` and `--version` options.
- Updated for compatibility with CouchDB release 0.7.0.

1.5.15 Version 0.1 (Sep 23, 2007)

- First public release.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index